Introduction to the Shake SDK

Extending Apple's Professional Applications

Session 300

These are confidential sessions—please refrain from filming or taking pictures.

Apple Applications

Shake SDK

Extending Apple's Professional Applications

Angus Taggart
Job Title

http://developer.apple.com/wwdc2005/

Course Objectives

- Introduce Shake
  - What the product does
  - Who uses it
  - Who develops for it
- High-level Overview of Shake's Architecture
  - Dependency graph concepts
  - Rendering Engine concepts
- C++ Plug-in Building Blocks
  - Key base classes
- Image Processing Plug-ins

Part 1: Introduction

- What is Shake??
- Customer Base
- 3rd Party Developer Community
- SDK Package

What Is Shake??

- Shake is Apple's high-end digital compositing and visual FX software solution
  - Unified 2D/3D compositing environment
  - Advance image processing
  - Editing and FX integration
  - Open, customizable API

Example FX Build-up Using Shake

Lord of the Rings, Courtesy WETA Digital

Customer Base

- Used extensively by the worlds top film visual FX companies
  - USA: ILM, Pixar, Dreamworks SKG, Tippet, Blue Sky ViFX
  - Europe: Cinesite, Moving Picture Company, Double-Negative, CFC-Framestore, Jim Henson's Creature Shop
  - New Zealand/Australia: WETA Digital, Animal Logic, Rising Sun Pictures
- Required courseware at major film schools worldwide
- Rapid adoption in the video post-production marketplace
  - Mac OS X native version (AltiVec optimizations)
  - Aggressive pricing

3rd Party Developer Community

The Foundry

Advant Edge

RE:Vision Effects

SFX

GenArts

The Pixel Farm

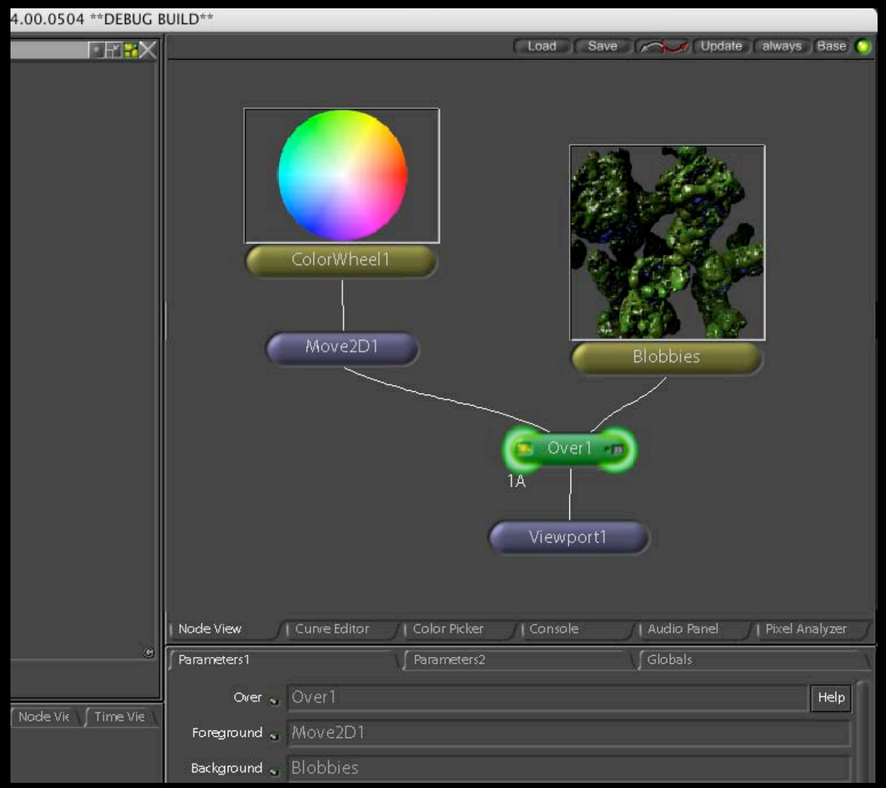
Algolith

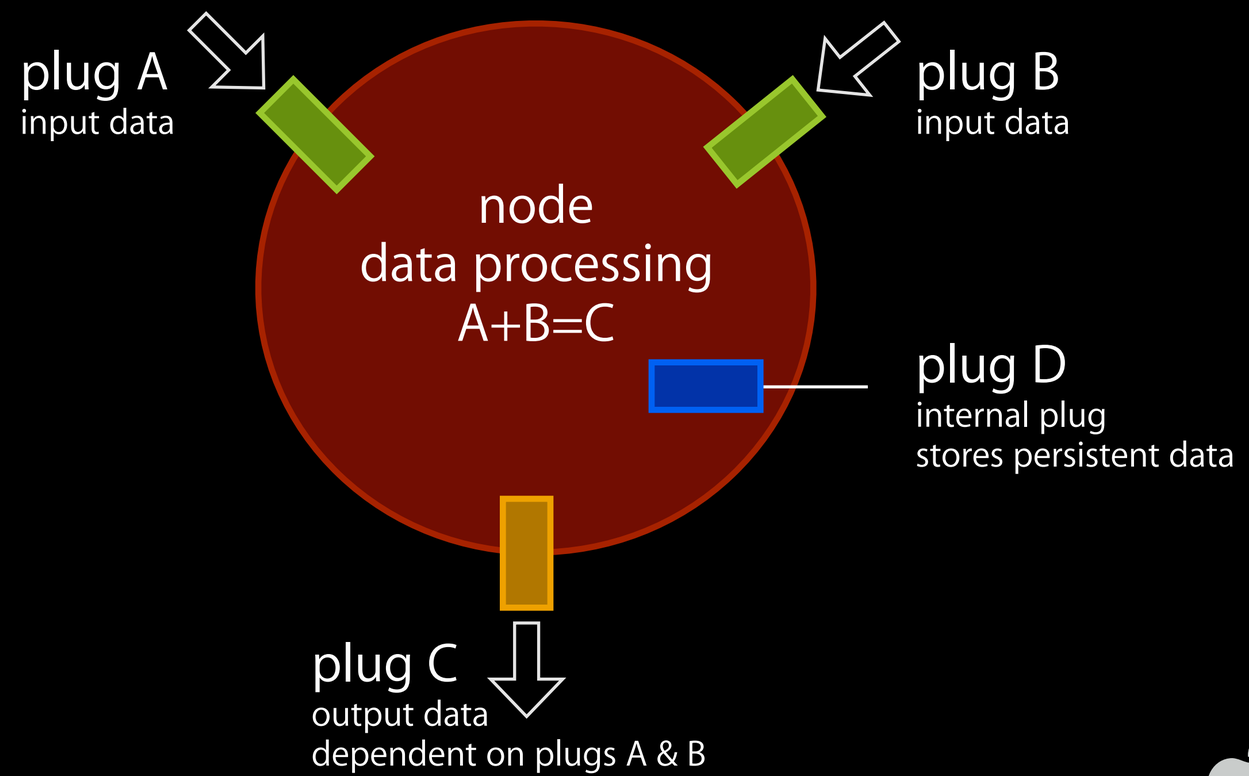
Part 2: Shake Architecture

- Dependency Graph Concepts
- Rendering Engine
- Composition Workspace

Dependency Graph Architecture (DGA)

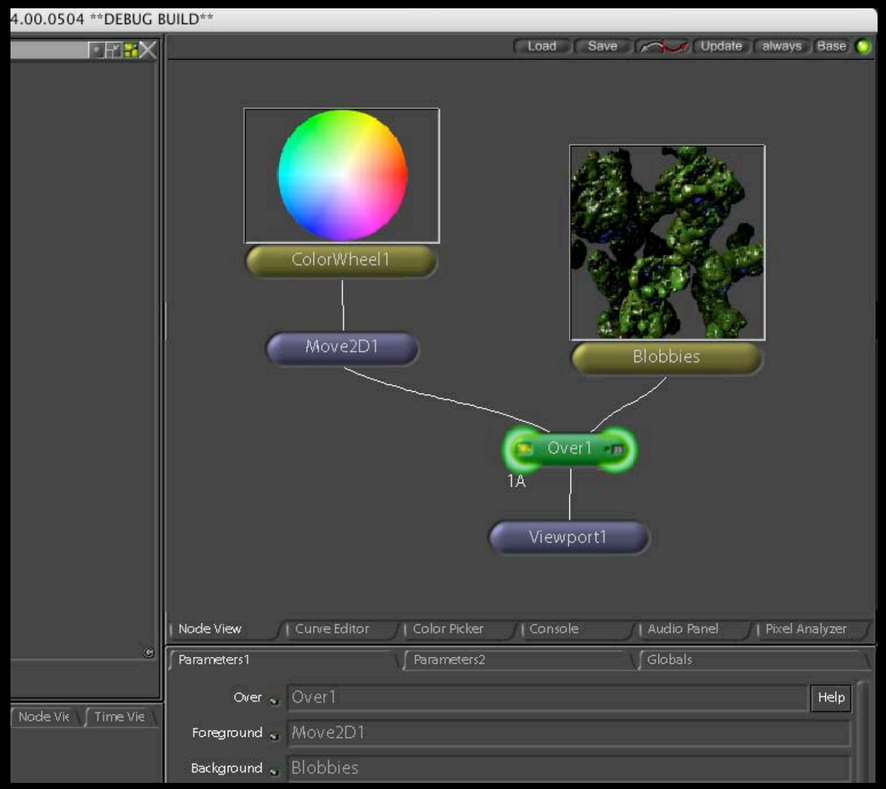
- Data processing occurs in Nodes
- Data flows in/out of nodes through Plugs

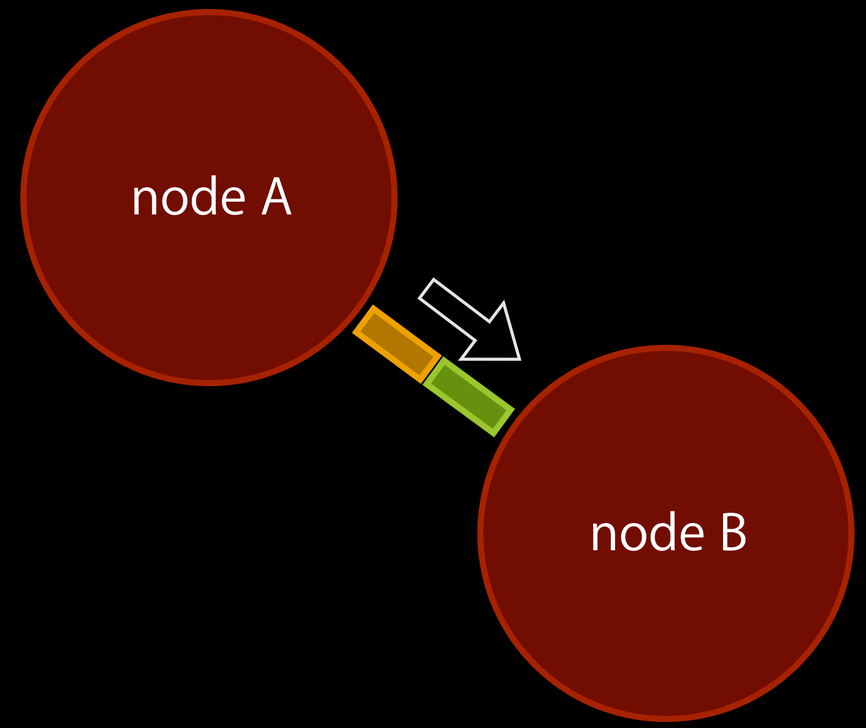




DGA: Nodes (Cont.)

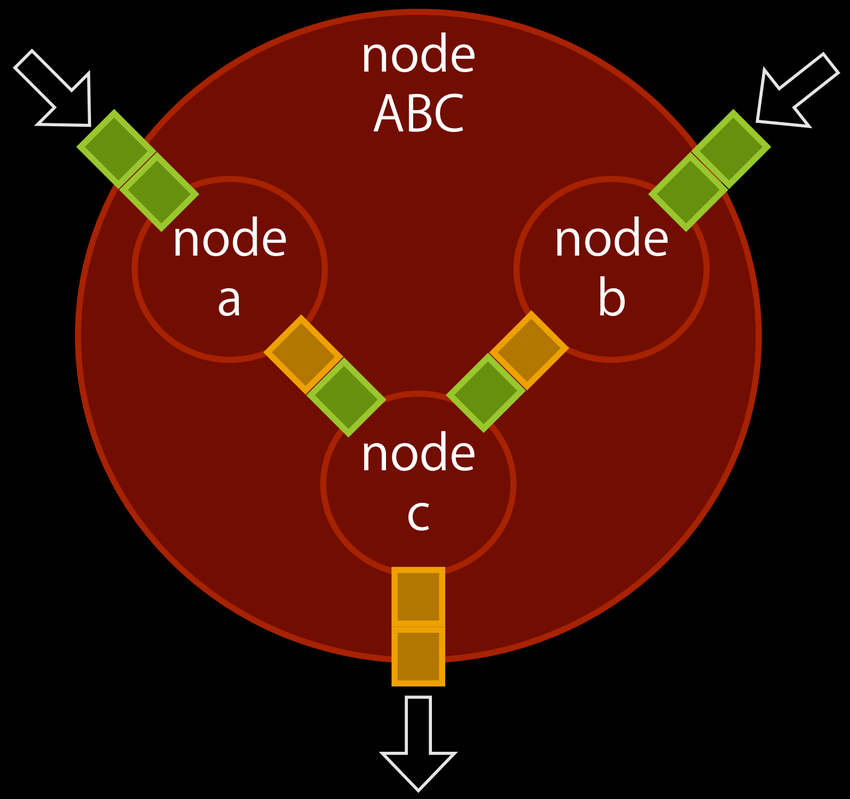
- Nodes are connected via plugs
  - Primary way to move data in a dependency graph
  - Plugs must be of the same data type
  - Connected plugs share a reference to the data





DGA: Nodes

- Support hierarchical parent/child relationships
  - Composite nodes incorporate one or more Atomic nodes
  - Enables more rapid creation of complex processing nodes by reuse of existing processing capabilities

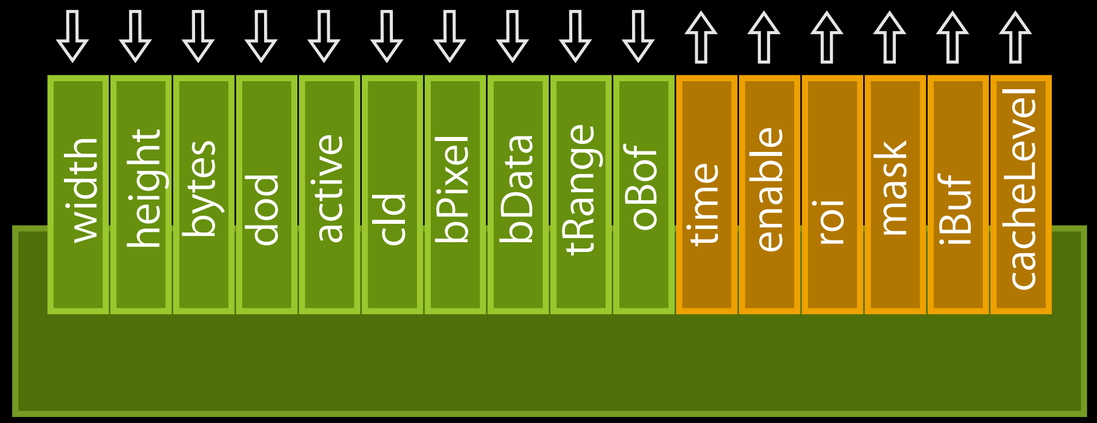


DGA: Plugs

- Must always be owned by a node
- Have directionality (in, out, both)
- Plug data types
  - int
  - float
  - double
  - string
  - pointer

DGA: Plugs (Cont.)

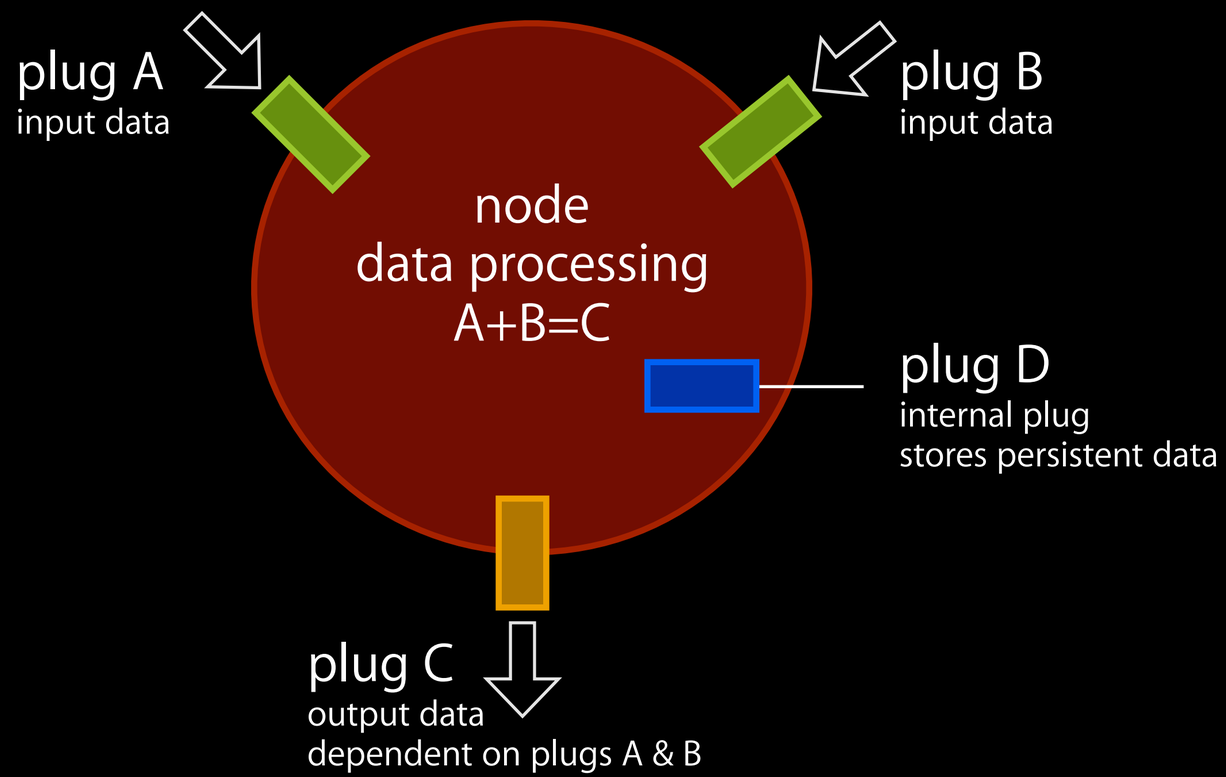
- Support hierarchical parent/child relationships
  - Composite plugs enable passing of complex data structures



Composite Plug with Bi-directional Children

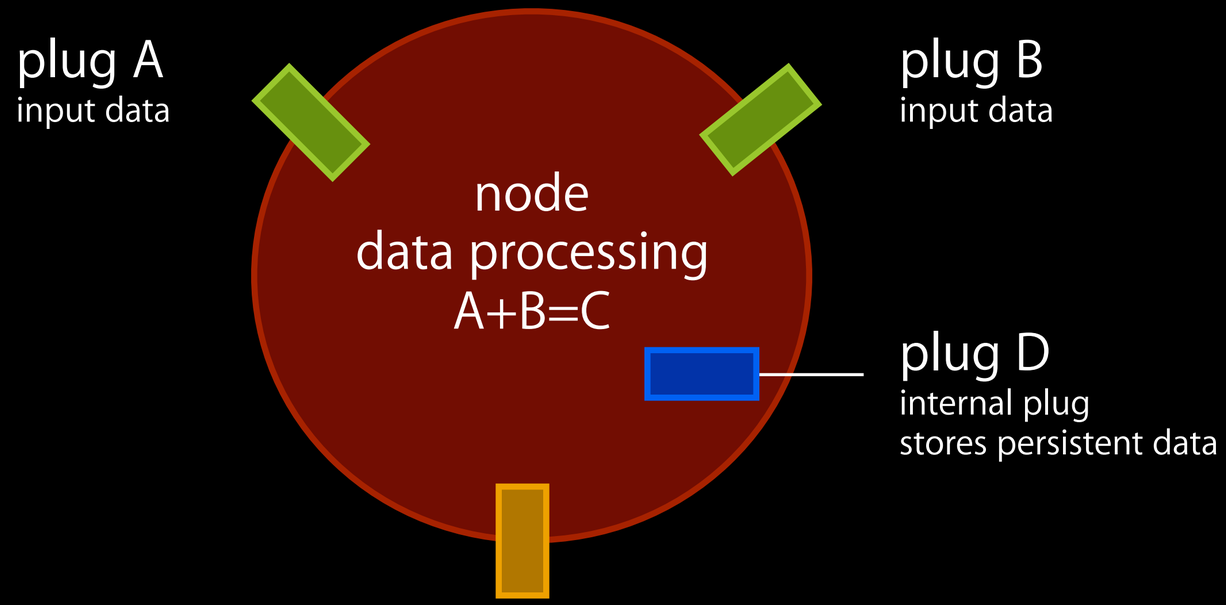
DGA: Plug Dependencies

- Must tell Shake which input plug(s) an output plug depends upon
- If values of input plug(s) that an output plug depends upon are "up-to-date", cached value of output plug will be used



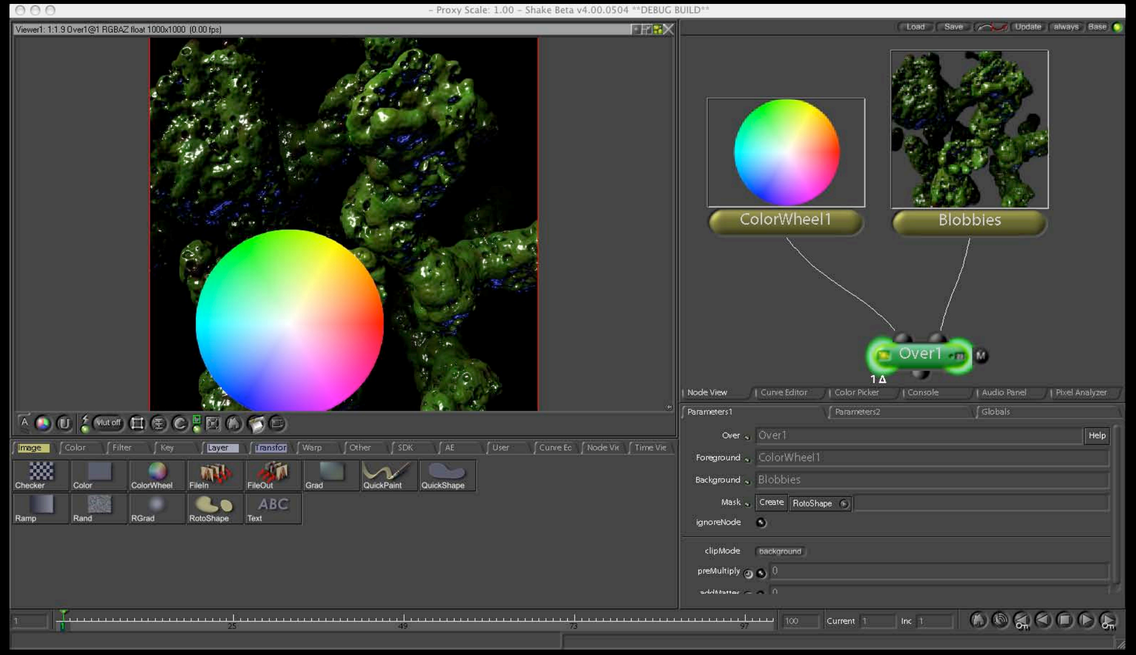
DGA: Lazy Evaluation

- Nodes need only perform processing when an output plug value is requested is by Shake



Rendering Engine

- Supports 8-bit, 16-bit (integer), and 32-bit float image data
- Resolution independent
- Multiple bit depths and image resolutions can be processed in the same Shake project

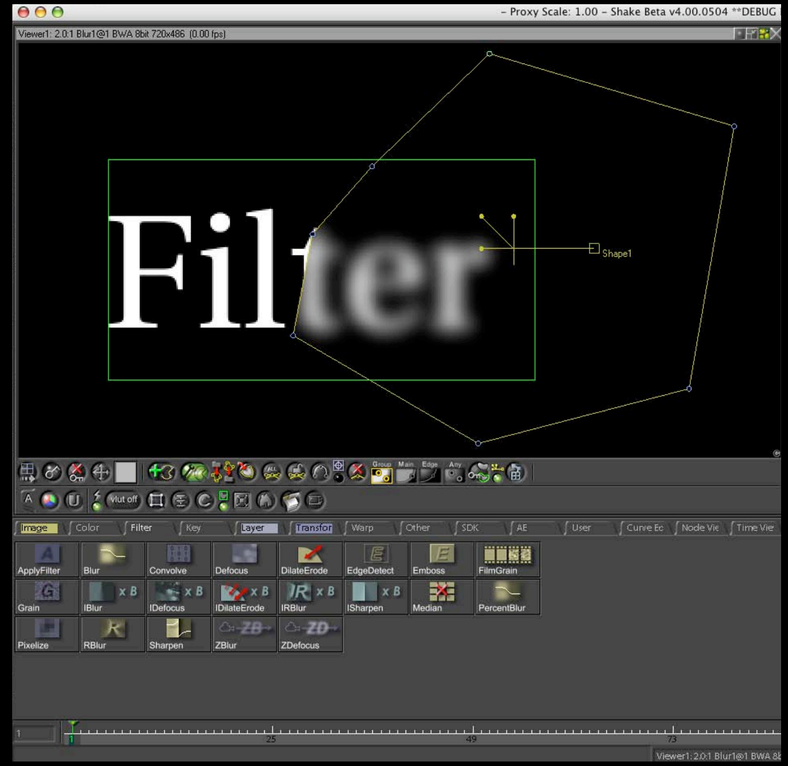


Rendering Engine

- Shake incorporates a hybrid block-scanline/tile-based rendering engine
  - In block-scanline mode, Shake allocates an image buffer comprised of one, or more, scanlines that are sized to fit the CPUs L2 cache
  - Images are rendered sequentially in scanline blocks from bottom-to-top
  - Significantly reduces Shake's memory footprint

Rendering Engine—Tile-Based Mode

- For image processing operations that require neighboring pixels (e.g. filters, warps), Shake will generate tiles from the input image on an "as requested" basis
- Image tiles are cached using a Least Recently Used (LRU) scheme for rendering efficiency

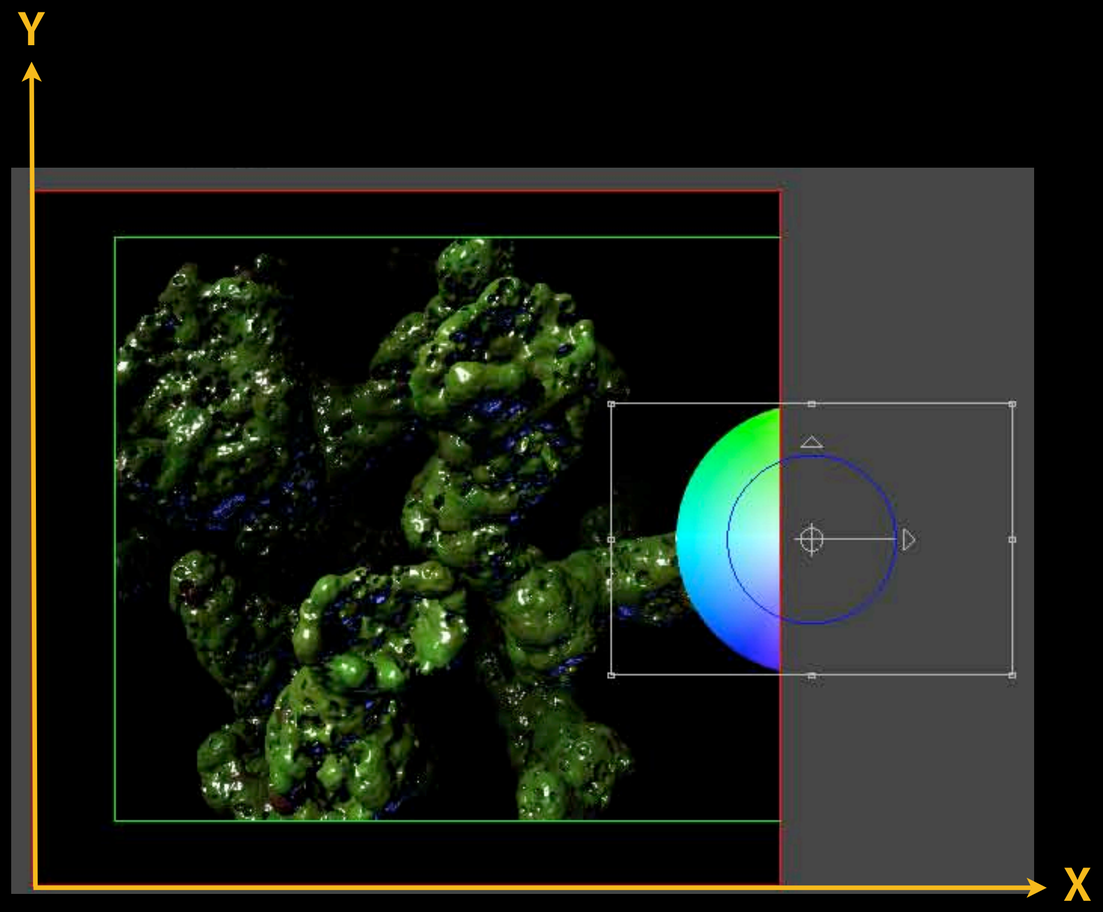


Rendering Engine—Efficiency Tip

- Plug-in developers should always attempt to implement image processing operations in a “block-scanline” friendly manner
- Requesting full-frame image buffers can adversely affect rendering performance by increasing Shake's memory footprint
- Remember that your node be used in several places within a Shake project and that Shake users regularly work with 2K and 4K float image data

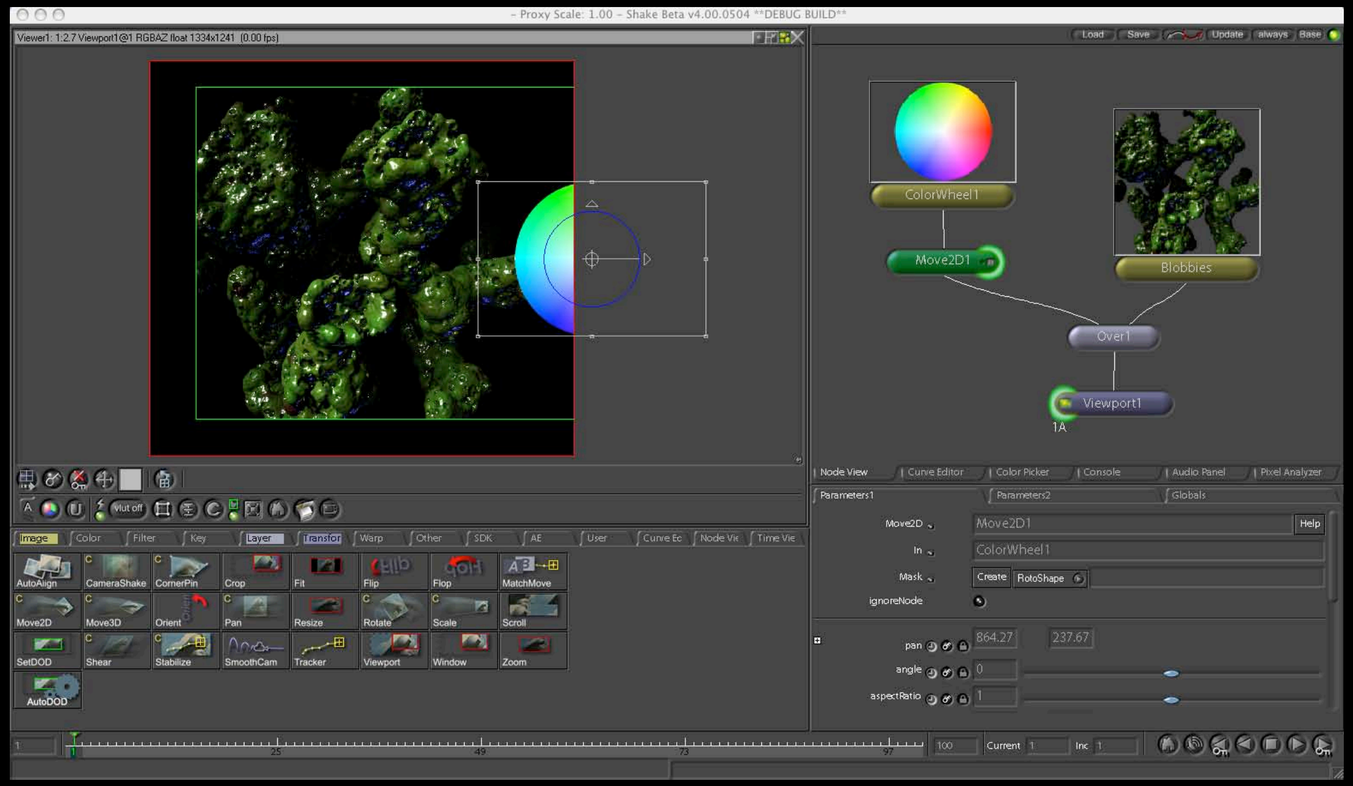
2D Coordinate System

- Origin of Shake's 2D Coordinate System is anchored to the lower-left hand corner of the Image Window



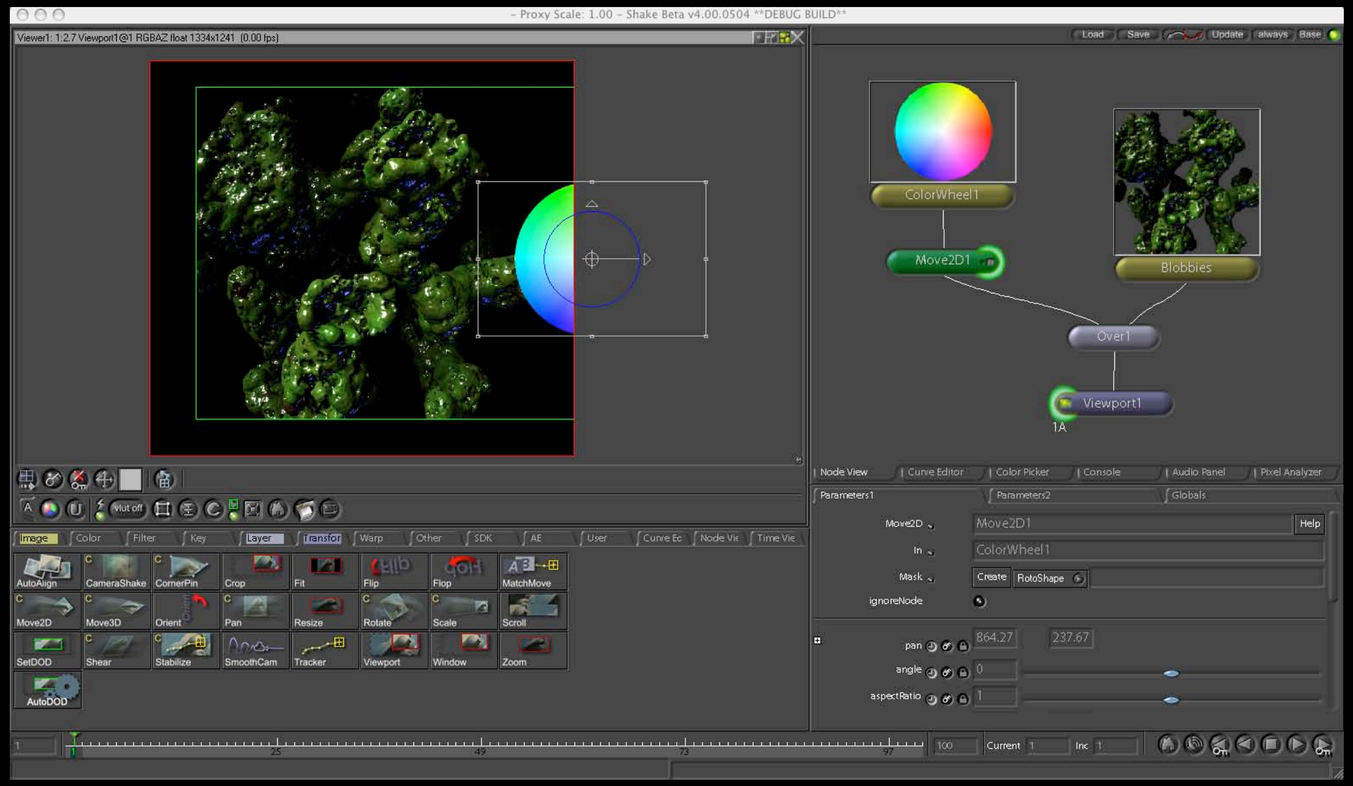
Composition Workspace

- Shake employs an Infinite Composition Workspace
  - There are not fixed boundaries where image content is clipped and lost
- The Background Pixel (bPixel) is defined the RGBA color of the composition's canvas



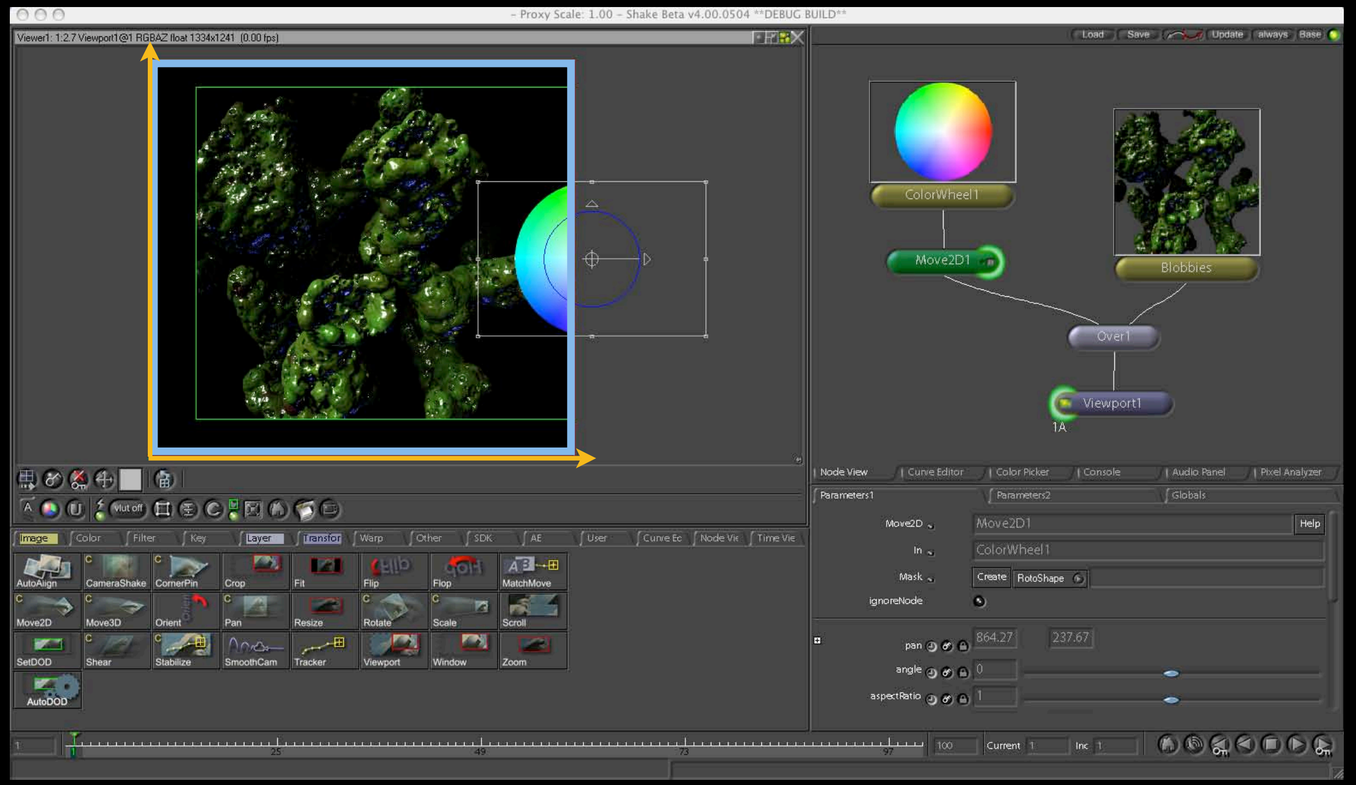
Composition Workspace (Cont.)

- Shake defines two bounding boxes to control where processing occurs
  - Domain of Definition (DOD): A bounding box that contains all non-background pixels
  - Region of Interest (ROI): A bounding box that that defines where processing should occur



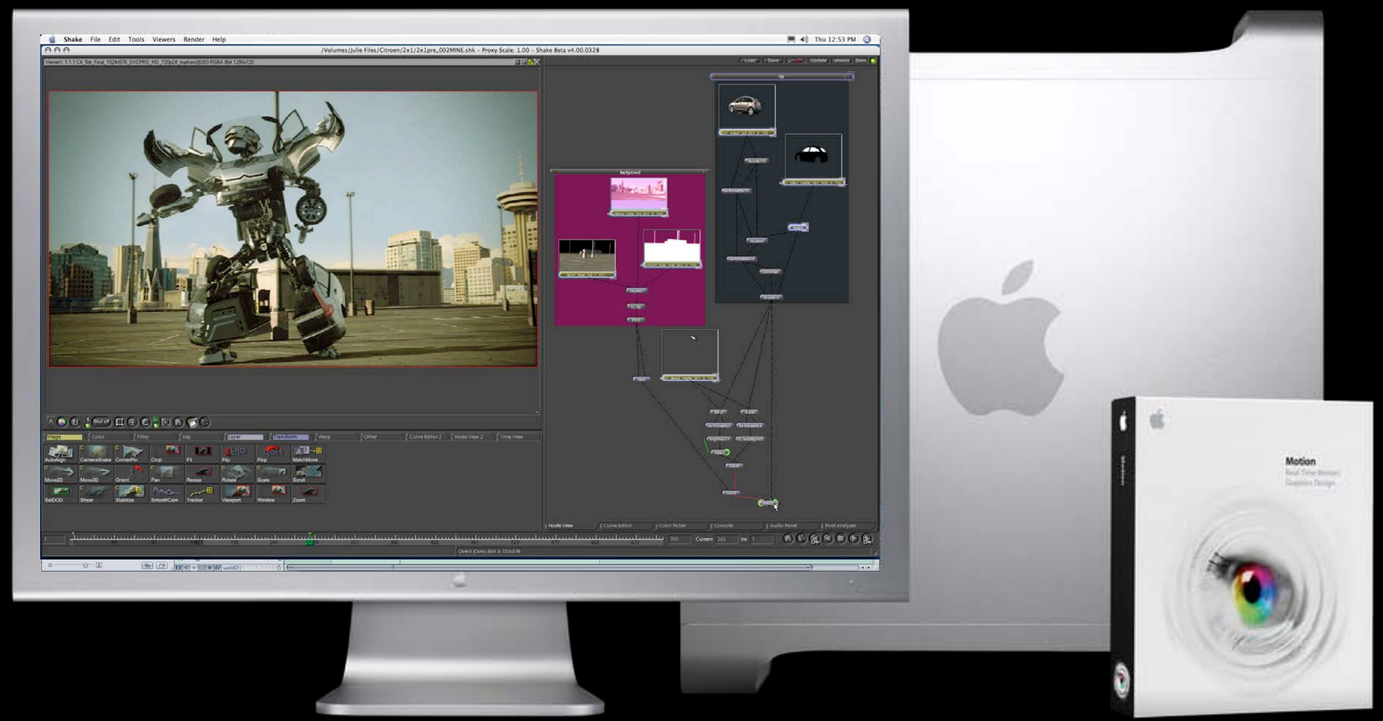
Composition Workspace (Cont.)

- Image processing node will only be asked to process in the region defined by the intersection of the ROI with the DOD



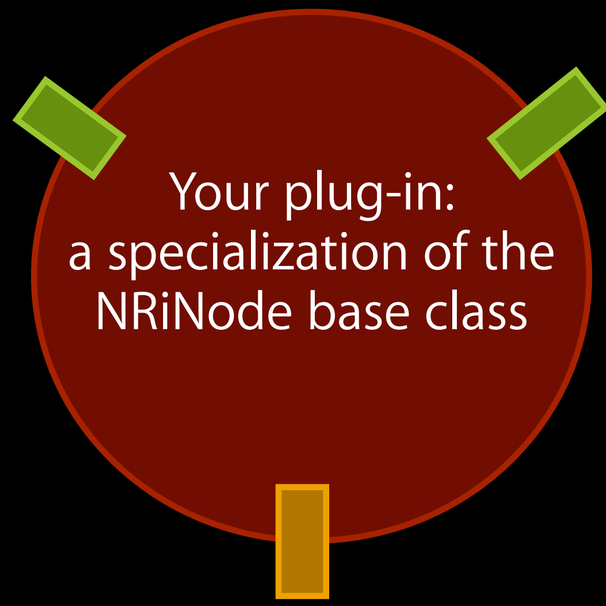
Part 3: Plug-in Basics

- Code Building Blocks
- Example Plug-in



Code Building Blocks

- Plug-ins are developed using C++
  - Nodes are C++ classes rooted at the NRiNode base class
  - Plugs are C++ classes rooted at the NRiPlug base class



- Plug-in developers rarely need to derive custom plugs

Code Building Blocks

The NRiNode dependency graph interface functions

NRiNode Function	Description
int eval(NRiPlug *p)	This function must be overridden to update the values of plugs that the node owns. A majority of a node's processing logic is triggered by calls to this function
int notify(NRiPlug *p)	This function can be overridden to receive notification when the value of a plug owned by the node changes. Nodes must set a NRiPlug::kNotify flag on plugs to receive
int bgnExec()	This function can be overridden to implement rendering setup tasks on a per-frame basis. This function is called before each frame is rendered and also for certain UI events

Code Building Blocks

The NRiNode dependency graph interface functions (Cont.)

NRiNode Member Function	Description
int endExec()	This function can be overridden to implement rendering cleanup tasks on a per-frame basis. This function is called before after frame is rendered
int serialize(NRiFile &fp, int mask)	This function can be overridden to modify NRiNode's default serialization of the input plugs that a node owns
int checkInterrupt() NOTE: utility function not a virtual function	This function can be called by a node in large processing blocks to check if the user has attempted to interrupted execution

Code Building Blocks

- Getting plug values
  - int NRiPlug::asInt()
  - float NRiPlug::asFloat()
  - double NRiPlug::asDouble()
  - void* NRiPlug::asPtr()
  - NRiName NRiPlug::asString()

- Note: To get a const char* from a string plug, call the NRiName::getString () member function

Code Building Blocks

- Setting plug values: NRiPlug::set*(value)*

  - Note: Conversions are performed e.g. setting an integer plug with a value of 1.0f will result in a conversion to 1. Also, the string "1.0" will be converted to 1.0f when setting a float plugs value

Code Building Blocks

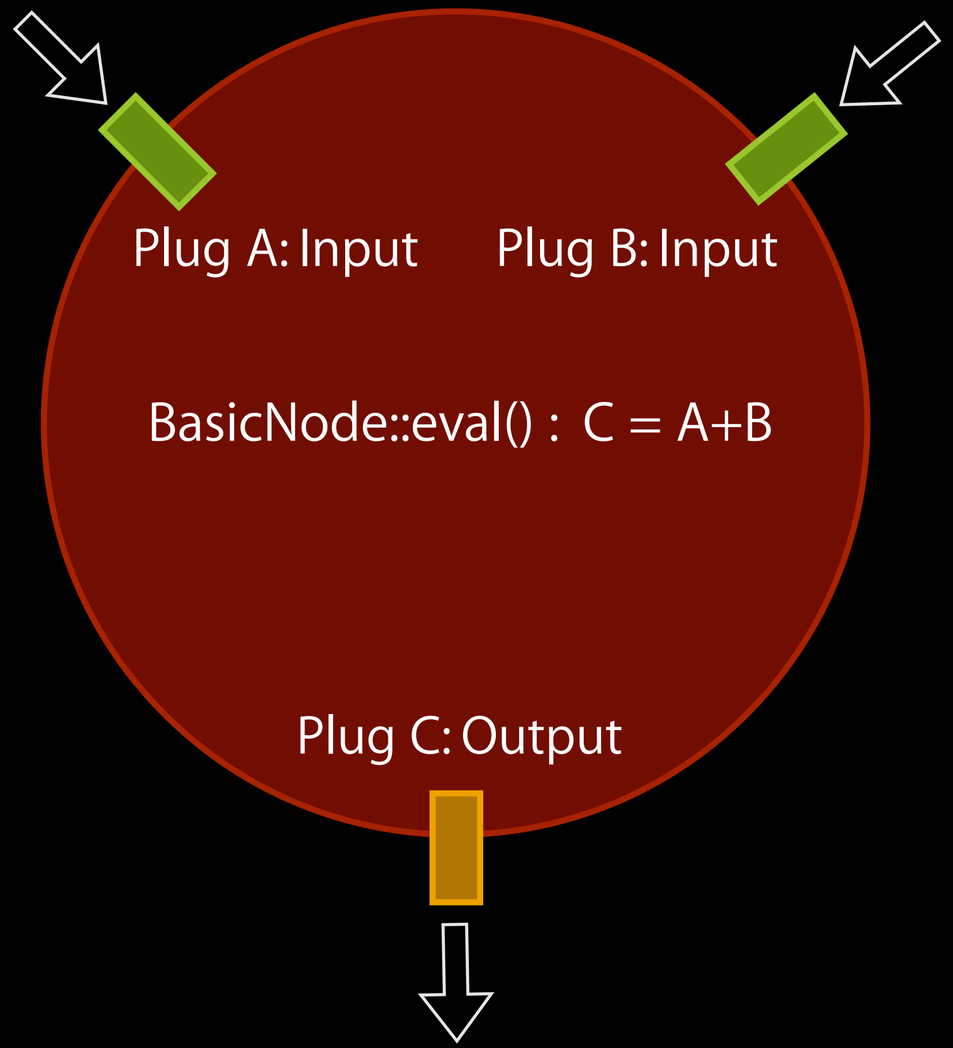
- Connecting Plugs: NRiPlug::connect(NRiPlug *p)
  - Connections can only be made between plugs of the same type
  - The plug that makes the connection gets its value from the plug to which it connects e.g. if pA->connect(pB), then pA's value = pB's value

Code Building Blocks

- Adding plugs to a node
  - NRiNode::addPlug(const NRiName &n, NRiId t, NRiPlug::IO io= NRiPlug::kIn, int internal = 0, int first = 0)
  - NRiId {kInt, kFloat, kDouble, kPtr, kString}
  - NRiPlug::IO {kIn, kOut, kInOut}
  - If the internal arg is 1, then the plug will not be displayed in the UI and will not be serialized
  - If the first arg is 1, then the plug will be positioned as the first plug in the node's list of plugs

Example Plug-in

BasicNode



Example Plug-in: Class Definition

```
class BasicNode : public NRiNode {
public:
    // Pointers to dynamically added member plugs.
    NRiPlug *_pA;
    NRiPlug *_pB;
    NRiPlug *_pC;

    BasicNode(); // Add plugs, register dependencies, etc.
    virtual ~BasicNode(); // Note: NRiNode base class deletes
                          // dynamically added member plugs

    // Dependency Graph interface functions
    virtual int eval(NRiPlug *p);

    NRiDeclareNodeName(BasicNode);
};
```

BasicNode Constructor

```
// Implement node setup logic in the constructor.
BasicNode::BasicNode()
{
    // Add plugs
    NRiPlug *_pA = addPlug("A", kInt, NRiPlug::kIn);
    NRiPlug *_pB = addPlug("B", kInt, NRiPlug::kIn);
    NRiPlug *_pC = addPlug("C", kInt, NRiPlug::kOut);

    // Register dependencies
    _pC->addDepenencies(_pA, _pB, 0);

    // Set initial values for input plugs
    _pA->set(1);
    _pB->set("4");
}
```

BasicNode eval()

```
// Update plug values.
int BasicNode::eval(NRiPlug *p)
{
    // Update the value of the output _pC plug
    if (p == _pC)
    {
        int a = _pA->asInt();
        int b = _pB->asInt();
        int c = a+b;
        p->set(c);
    }
    return NRiNode::eval(p);
}
```

Part 4: Image Processing Nodes

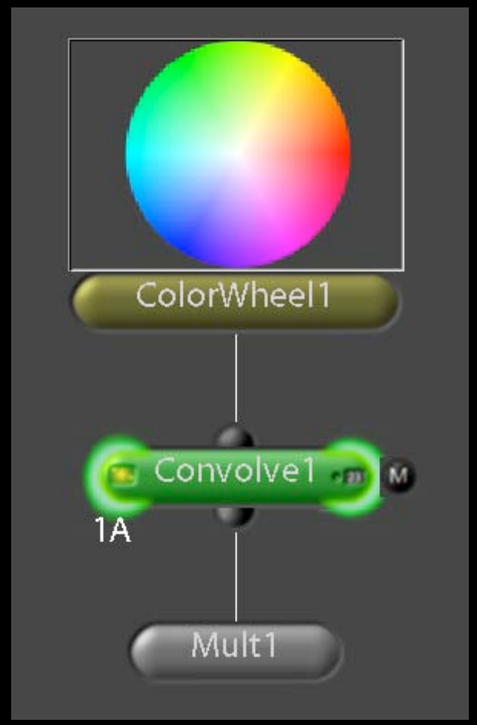
- Plug-in Types
- The Image Plug
- Image Processing Base Classes
- “C” Language Creator Function for Macro Binding

Plug-in Types

- Image Processing
- On-Screen Controls (OSCs)
- Spline Data Import/Export
- Render Queue Interface
- Widgets/UI Elements
- Custom File Format Readers/Writers

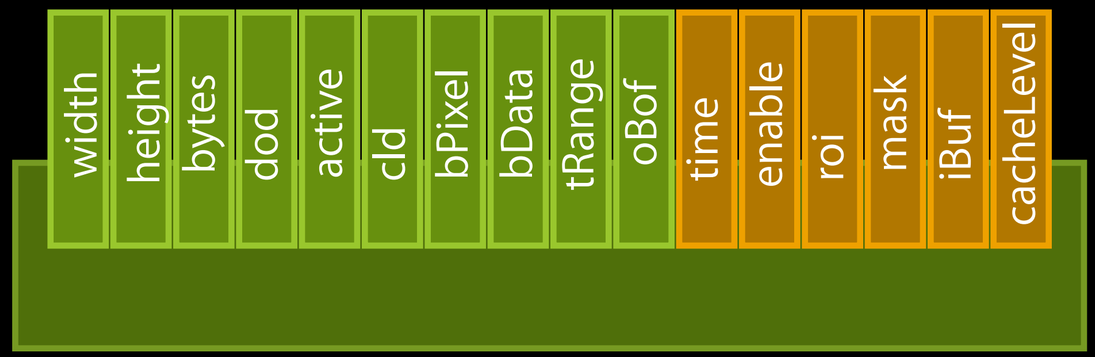
The Image Plug

- The Image Plug is the primary connection between Shake's image processing nodes
- The Image Plug carries all the information and data associated with the image buffer into and out of image processing nodes



The Image Plug—Structure

- The Image Plug is a composite plug that contains 16 child plugs
- Of the 16 child plugs, 10 of them carry data down the processing tree, 6 of them carry data up the processing tree
- Defined by the NRilPlug class



The Image Plug—Downstream Children

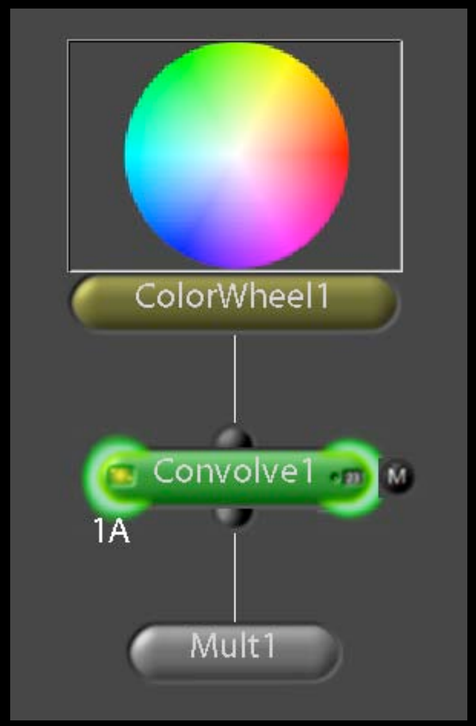
Child Plug	Data Type	Description
width	int	width of the image
height	int	height of the image
bytes	int	bit depth (1=8bit,2=16bit,4=32bit)
dod	NRiIRect*	domain of definition
active	int	active color channels
cId	NRiName	cache ID—must be unique string that represents the up-to-date state of a node
bPixel	NRiBPixel*	background pixel
bData	NRiBData*	blind data
tRange	NRiTimeRange*	valid time range for image data
oBuf	NRiIBuf*	pointer to a structure containing the processed image buffer

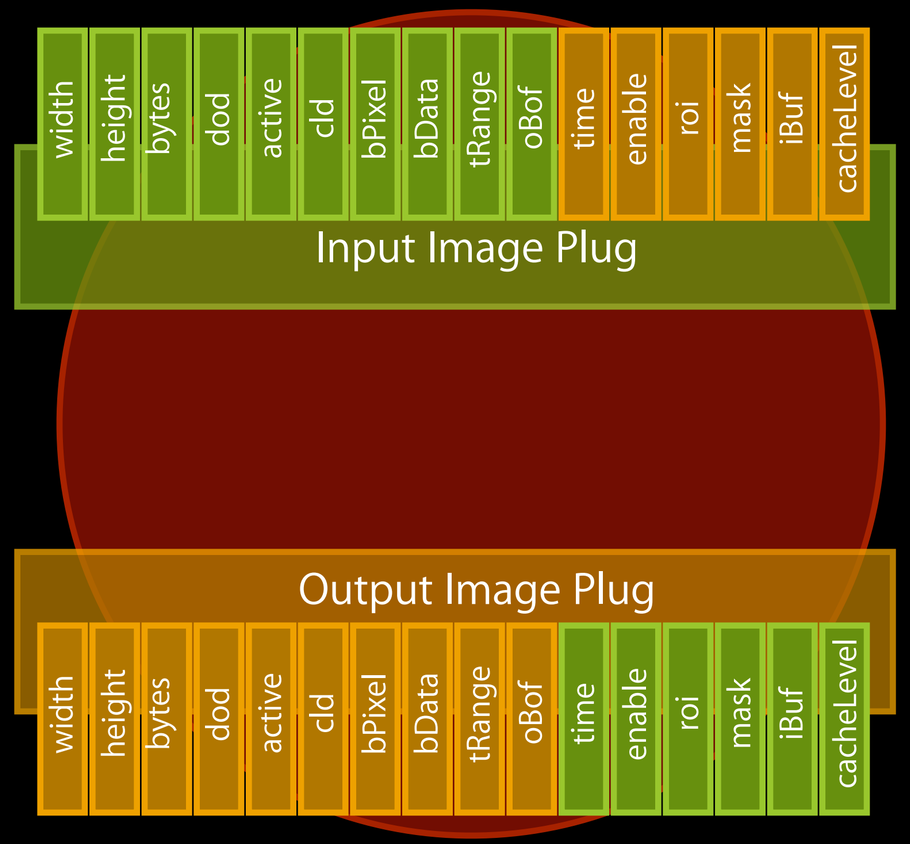
The Image Plug—Upstream Children

Child Plug	Data Type	Description
time	float	frame time
enable	int	enabled state of the node
roi	NRiIRect*	region of interest
mask	int	requested color channels
iBuf	NRiIBuf*	pointer to a structure containing the un-processed image buffer
cacheLevel	int	indicator of cache worthiness

The Image Plug—Polarity

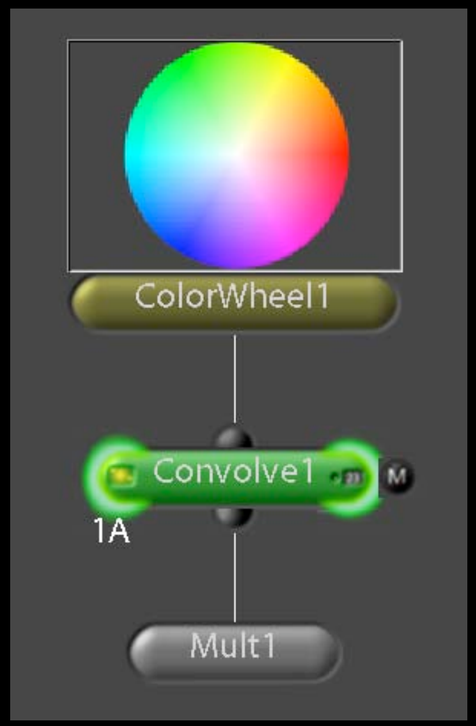
- The child plug's IO polarity flips depending on whether the image plug is an input to a node or an output to a node

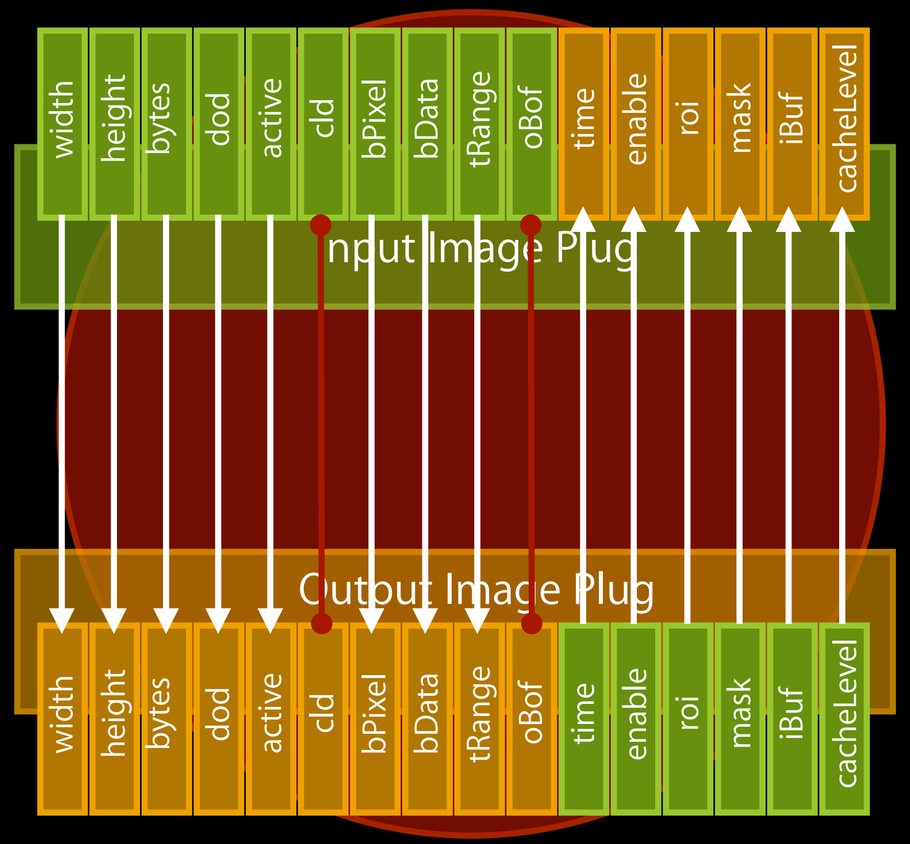




The Image Plug—Evaluation Logic

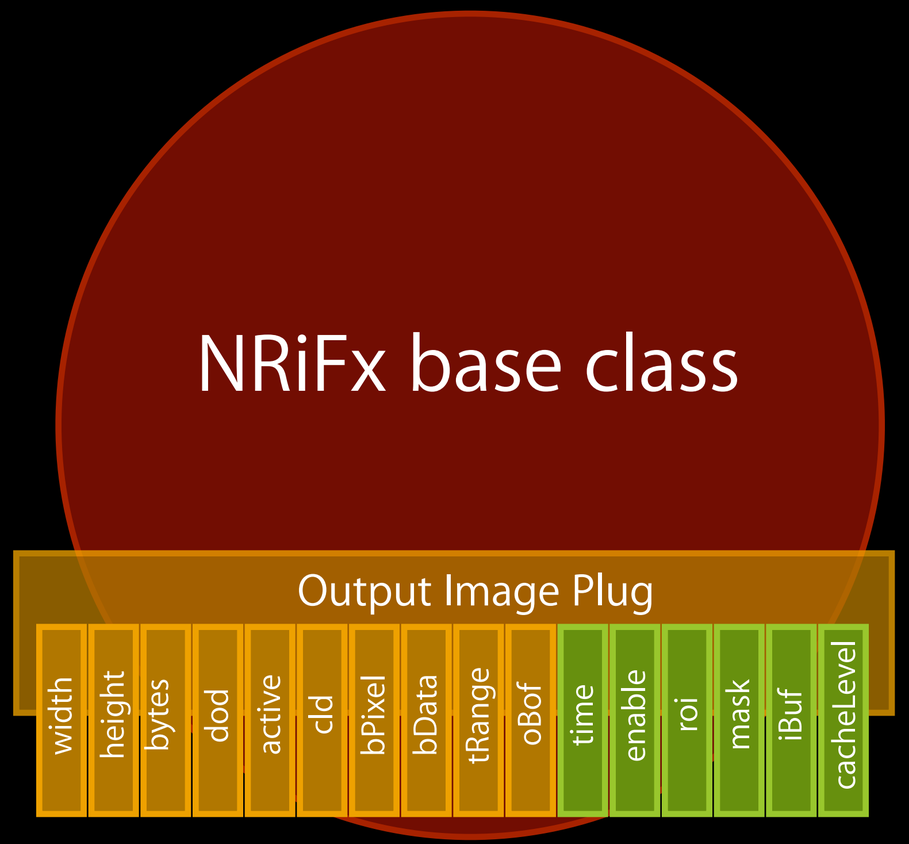
- Image processing is performed by evaluating the un-connected output child plugs
- At a minimum, the cacheId and oBuf plugs should be un-connected





Shake Image Processing Base Classes

- Implement functionality for specific types of image processing operations
- Greatly simplify developing Shake plug-ins
- Are rooted at the NRiFx base class, which defines an image processing node with an output image plug



NRiMonadic and NRiCMonadic

Class (Parent class)	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiMonadic (NRiFx)	Base class for non-computationally intensive single input image processing nodes	in	out	no
NRiCMonadic (NRiFx)	Recommended base class for most single input image processing nodes Adds an embedded caching node before the out image plug	in	cacheOut	yes

NRiDiadic

Class (Parent class)	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiDiadic (NRiCMonadic)	Base class for layering nodes that require two input images Allocates the image buffer for the second input image and conforms the resolution, bit depth, and active color channels of the two inputs Defines the pure virtual composite() function that derived classes must override to implement image processing operations. The NRiDiadic base class passes the two input image buffers into this function	in1p in2p	cacheOut	yes

NRiLookup

Class (Parent class) sdk example	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiLookup (NRiShortcut)	Base class for look-up table color operators. Defines the pure virtual fofx() function that derived classes must override to implement logic to build the look-up table. The NRiLookup base class builds the look-up table by calling the derived class' fofx() function with an input color value and the derived class must compute the modified color value	—	—	yes

NRiMLines

Class (Parent class) sdk example	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiMLines (NRiCMonadic)	A base class for image processing nodes that require neighboring pixel data to perform image processing operations e.g. filters Allows derived classes to specify the region about the current pixel that they need for processing Defines the pure virtual processLine() function that derived classes must override to implement image processing operations	in	cacheOut	yes

NRiWarp

Class (Parent class) sdk example	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiWarp (NRiTCache)	A base class for image processing nodes that warp pixel locations without modifying RGB values Defines the pure virtual warp() function that derived classes must override to implement warping logic. The NRiWarp base class calls the derived class' warp() function with a set of pixel locations and the derived class must pass back the warped pixel locations(s)	in	cacheOut	yes + tile cache

NRiNadic

Class (Parent class) sdk example	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRiNadic (NRiMonadic)	Base class for image processing nodes that require more than one input image buffer Defines the virtual fillOutputBuffer() function that derived classes must override to implement processing logic. All input image buffers will be allocated, filled, and conformed when this function is called Provides an optional full-frame image buffer processing mode	pIn[]	pOut	yes

NRillnput

Class (Parent class) sdk example	Description	Input Processing IPlug(s)	Output Processing IPlug	Image Caching
NRillnput (NRiFx) input.cpp	A base class for image processing nodes that will generate image that is dependent on frame time This class provides functionality for defining valid time ranges, defining behavior outside of valid time ranges, and computing a localized frame time when an image sequence has been shifted in time	—	out	no

More Information

http://developer.apple.com/wwdc2005/

Documentation, Sample Code, and Other Resources

http://developer.apple.com/wwdc2005/

Who to Contact

Patrick Collins
Technology Manager
patrick@apple.com

Angus Taggart
Shake SDK
angus@apple.com

Q&A

Patrick Collins
Technology Manager
patrick@apple.com

http://developer.apple.com/wwdc2005/

WWDC2005

Worldwide Developers Conference